Context-Sensitive Detection of Information Exposure Bugs with Symbolic Execution

Paul Muntean, Claudia Eckert and Andreas Ibing
Technical University Munich
Chair for IT Security (I20)
Boltzmannstraße 3
85748 Garching, Germany
{paul.muntean, claudia.eckert}@in.tum.de, ibing@sec.in.tum.de

ABSTRACT

Static analysis tools used for detecting information exposure bugs can help software engineers detecting bugs without introducing run-time overhead. Such tools can make the detection of information-flow bugs faster and cheaper without having to provide user input in order to trigger the bug detection. In this paper we present a bug-detection tool for detecting information exposure bugs in C/C++ programs. Our tool is context-sensitive and uses static code analysis for bug detection. We developed our bug finding tool as a Eclipse plug-in in order to easily integrate it in software development work flows. The bug reports provide user friendly visualizations that can be easily traced back to the location where the bug was detected. We discuss one static analysis approach for detecting information exposure bugs and relate briefly the usability of our bug testing tool to empirical research. We conducted an empirical evaluation based on 90 test programs which were selected from the Juliet test suite for C/C++ code. We reached a true-positive coverage of 94.4% in 121 seconds for 90 test programs having a total of 12589 source code lines.

Categories and Subject Descriptors
D.2.2 [**Design Tools and Techniques**]: Programmer workbench; D.2.5 [**Testing and Debugging**]: Symbolic execution; D.2.5 [Testing and Debugging]: Testing tools; D.2.6 [**Programming Environments**]: Integrated environments;

General Terms
Security, Verification, Experimentation

Keywords
Software bugs, software testing, information-flow, static taint analysis, integrated development environment

1. INTRODUCTION
Information exposure weaknesses are a type of Information Flow (IF) weaknesses. IF weaknesses represent one type of software weakness, which can exist in the software without directly breaking the code but rather offering useful information to an attacker who could exploit IF leakages [1]. These types of software bugs can lie dormant in an application for a long time period without being detected and can cause huge harm [1]. According to Common Weakness Enumeration (CWE) CWE-200 (the parent weakness class of the test programs used in this paper) [2] Information Exposure (IE) is the "intentional or unintentional disclosure of information to an actor that is not explicitly authorized to have access to that information". IE vulnerabilities are a subtype of IF vulnerabilities. As of 2007 IE leakages rank 6th in the AOWASP top ten list [3] and as of 2010 rank 7th according to VERACODE mobile app top ten list [4]. We argue that software should be thoroughly tested before it is released in order to detect potential exploitable IF vulnerabilities.

The process of software testing accounts for more than 50% of the whole effort during software engineering projects according to [5] and [6]. Detecting software bugs, which cause information exposure vulnerabilities is crucial because potential exploitation possibilities should be removed from source code before release. Software weaknesses are hard to detect and can cause information leaks which attackers can exploit. By building Control Flow Graphs (CFG) which describe possible execution paths and tracking taint data as it "moves" along the path nodes guarantees high path coverage.

Many static analysis approaches are very promising but still have to be applied to security scenarios. At the same time a relative high number of tool vendors (Microsoft, IBM, Coverity, klocWork, Infosys, Cognizant, Hexaware) start to address the need for static analysis into mainstream tools. Some example tools are ESP [8] a large scale property verification approach, model checkers as SLAM [9] and BLAST [10] which use predicate abstraction to examine program safety properties. FindBugs [11] a lightweight byte code checker based on predefined bug patterns.

Triggering IF bugs is not a trivial job and can be addressed using dynamic analysis, static analysis or hybrid approaches. Dynamic analysis introduces computing overheads and it cannot guarantee that all possible execution paths are exercised. Where as static execution provides all potentially execution paths but needs some heuristic for selecting only the relevant paths. Also it is relevant to select only reachable paths, which can be determined with the help of an SMT solver. The mathematical expressions provided to the SMT solver often are blown up in size and can get very complex [12].

IF vulnerabilities can be addressed by dynamic analysis [14], [15], [16] static analysis [17], [18], [48], [19], [20] and hybrid approaches which combine static and dynamic mechanisms [21]. Tracking taint variables through the program execution is key to detect IE weaknesses, which are a type of IF vulnerabilities. IF controls focus on preventing leaks from confidential (or *high*) to output (or *low*) data. The desired baseline policy is *noninterference* [22] that demands that there is no dependence of public outputs on confidential inputs. There are two types of IF variants, which can be taken into account when dealing with variable interference. Information is passed from right-hand side to left-hand in an assignment through an *explicit flow*. Assume variables *confidential* and *output* have high and low security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

InnoSWDev'14, November 16, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3226-2/14/11...$15.00
http://dx.doi.org/10.1145/2666581.2666591

levels, respectively. For example, *output := confidential* exhibits an explicit flow from confidential to output. Information is passed via control-flow structure in an *implicit flow*. For example, if *confidential* then *output* := *true* else *output* := *false* has an implicit flow. The value of the *output* variable depends on the confidential variable. We will call a conditional or a loop *high* if its guard involves a high variable. Information-flow control is concerned with preventing explicit and implicit flows in order to guarantee non-interference.

One possibility to prevent explicit and implicit flows is by using purely static Denning-style enforcement [23]. Each assignment is checked if it fulfills the following conditions: the level of the assigned variable must be high when there is a variable on the right-hand side of the assignment (*tracking explicit flows*) or in case the assignment appears inside of a high conditional or loop (*tracking implicit flows*). This mechanism guarantees that no low computation occurs in the branches of high conditionals and loops [24]. Another possibility is through dynamic enforcement, which is based on dynamic security checks similar to the ones done by static analysis. Whenever there is a high variable on the right hand side in an assignment (tracking explicit flows) or the assignment appears inside a high conditional or while loop (tracking implicit flows) then the assignment is only allowed when the assigned variable is high. This mechanism dynamically keeps a simple invariant of no assignment to low variables in high context [24].

We have chosen the C programming language because it is widely used for developing embedded systems ranking currently first position in the TIOBE [25], Langpop [26] rankings of programming languages and second position in the IEEE Spectrum top 10 of most used programming languages [27]. We argue that embedded software should be tested more thoroughly and bug detectors should be integrated as early as possible in the software development cycle. We think that integrating bug-finding functionalities in an IDE will help to detect the IF bugs early in the development process of software systems. The Eclipse IDE is the most used Java IDE in the industry [28]. By designing an Information Exposure Checker (IEC), which can run in different running modes, we think that we can increase the productivity of the code debugging process. The IEC can detect bugs during run-time of the Eclipse IDE and it offers two main advantages. First, it offers the possibility of detecting IE bugs during development. Second, we get a high level of integration between the IDE and the bug detection mechanisms.

The goal of our research is to develop a tool for detecting IF exposure bugs using static analysis. The tool should use context-sensitive analysis and should rely on a Satisfiable Modulo Theories (SMT) [29], [30] solver. In summary we make the following contributions:

- We developed an IE detection tool capable to detect information exposure bugs fully automated using SMT-lib 2.0 [55].
- Inter-procedural, path-sensitive analysis and context-sensitive analysis was used to detect the IE bugs.
- We propose a new method to define sinks, sources and taint confidential symbolic variables by defining function models.
- We defined an easy method to add new checkers into the Static Analysis Engine (SAE) [31] by adding the required function models for the sinks, sources and tainting confidential variables.
- The SAE statement processor was extended to support explicit information flow propagation of symbolic variables.
- We designed our checker as an eclipse plug-in which can be run in different modes as presented in Fig. 8b.

2. MOTIVATION

The detection of information exposure bugs is based on finding the locations in source code where sensitive information is about to leave a trust-boundary. An attacker could exploit this IE vulnerability if this information is leaked to the outside of the system. This may contain sensitive information about a remote server or other secret resources. We want to build a tool capable to detect IE vulnerabilities.

```
0.   void CWE526_bad(){
1.    if (staticFive == 5){
2.     /*FLAW:environment variable exposed*/
3.      printLine(getenv("PATH"));
4.     }
5.   }
```

Figure 1a. CWE-526 test programs source, after [7].

```
0.   void printLine (const char *line){
1.    if(line != NULL){
2.       printf("%s\n", line);
3.     }
4.   }
```

Figure 1b. CWE-526 test programs sink, after [7].

Fig. 1a and Fig. 1b present a information exposure scenario between the source, Fig. 1a, line 3 and the sink Fig 1b, line 2. These code snippets are contained in the CWE-526 test case available in [7]. On line 3 in Fig. 1a the system `PATH` variable is sent to the `printf()` sink located on line 2 in Fig. 1b. The `printLine()` function presented in Fig. 1a is a wrapper for the C `printf()` function and it is contained in another C file. The trust-boundary is represented in this case by the `printf()` sink function. For the `getenv()` we define a function model where we set the return value of it to be confidential. The return value of the `getenv()` will be propagated using static execution and explicit IF. When a confidential symbolic variable is about to leave the sink `printf()` the interpreter will be notified. This represents the bug triggering condition used by our IE checker.

2.1 Challenges and Design Requirements

We formulated our research challenges as two questions: It is possible to successfully use static analysis and the SAE engine to develop an IF checker for detecting IE bugs? What are the performance increases in comparison to related research work? The solutions to our challenges will be presented throughout the sections 3, 4 and 6.

The IE checker should find IE vulnerabilities in the test cases Information Exposure Through Environment Variables (CWE-526), Information Exposure through Debug Log Files (CWE-534) and Information Exposure through Shell Error Message (CWE-535) obtained from the Juliet test suite [7]. The logic for detecting each type of vulnerability should be contained in one checkers class, which should be attached to the interpreter. The interpreter should trigger the checker at a possible bug location, i.e., output through a trust boundary. IF checker requirements:

- The IEC should track IF's and detect potential IE vulnerabilities.
- The IEC should be capable of checking test programs composed of more than one C/C++ file.
- Trust-boundaries should be defined using SAE [31].
- Confidential variables should be propagated based on explicit IF.

- The IEC should indicate the location in the test program file where the vulnerability was detected by using the mark-up label available in the Codan API [34].

The remainder of the paper is organized as follows. Section 3 presents the architecture of our IE checker. Section 4 contains the implementation of our checker. Section 5 contains the IE checker tool demo. Section 6 contains empirical results. Section 7 contains related work and section 8 is devoted to conclusions and future work.

3. ARCHITECTURE

Our IE checker is based on the static analysis engine [31] which is used for C/C++ source code analysis and has in the back-end the MathSat [32] SMT solver. The steps needed to extend the SAE are briefly explained. A new function model is created for each sink and source and added to the environment models package, which contains models for all potential sources and sinks present in the selected test cases. Function models are used to model sinks, sources and other types of trust-boundaries. At the same time function model are used to notify the interpreter when a previously tagged variable is about to pass through a trust-boundary. The interpreter will be notified by sending it a potential tagged symbolic variable. Afterward all the currently attached checkers will be notified by sending the tagged symbolic variable to them. Inside the checker class it is checked if the variable is confidential, sensitive, etc. If the check if positive then a bug report will be issued.

3.1 Static Analysis Engine Architecture

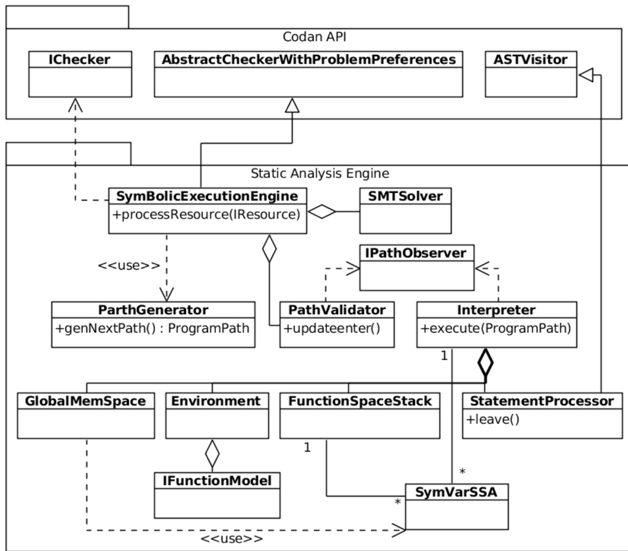

Figure 2. SAE architecture.

The function models contain a tainted symbolic variable with a confidential label assigned to it. The symbolic variable will be propagated along a path. The interpreter will be notified when passing over a sink. The sink notifies the interpreter by sending a symbolic variable, which could be confidential or not. The interpreter calls each previously attached checker. The symbolic variable is checked whether or not it is confidential by the checker. If the variable is confidential then a bug report will be issued. If additional logic for checking other relevant conditions is needed then this can be added in the IE checker class. The architecture of the used static analysis engine is presented in Fig. 2. A more detailed explanation of the main classes contained in the SAE can be seen in the paper [31]. The reused Codan API interfaces and classes are presented in the Codan API package shown in Fig. 2. The interface `IChecker` adds to the implementing class the possibility to work with project resources like: projects, files, etc.

The `AbstractCheckerWithProblemPreferences` class extends the class `AbstractChecker`. It contains methods for defining the run-time settings for the checker class. Checkers can generate several types of outputs. Each checker preference settings can be defined individually. The abstract `ASTVisitor` class is a Codan base class, which is extended by all visitor classes that need to traverse the nodes of an AST. The `ASTVisitor` implements the visitor design pattern. The `visit()` methods implement a top-down traversal and the `leave()` methods implement a bottom-up traversal of a C statement represented as an AST.

3.2 Information Exposure Checker Architecture

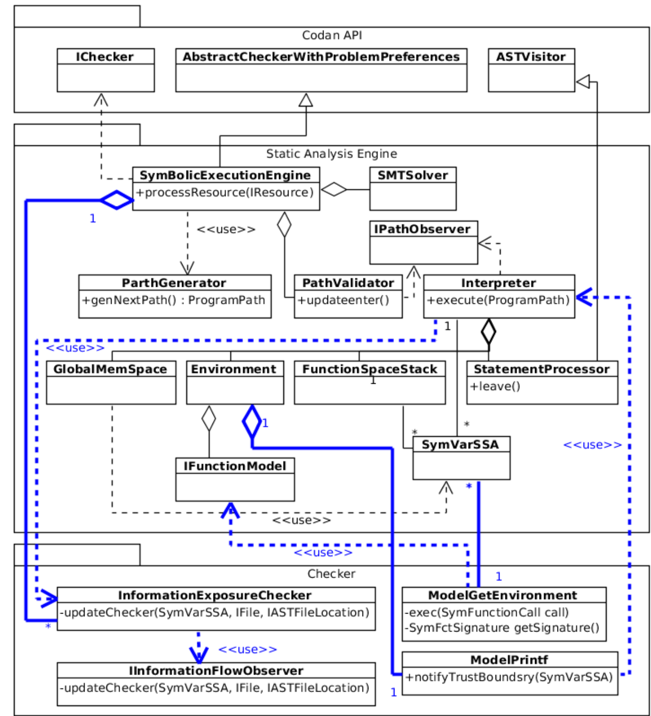

Figure 3. The IF checker architecture.

The blue lines in Fig. 3 indicate all the dependencies between the SAE presented in Fig. 2 and our IE checker. Implementation details for the `InformationExposureChecker` class (IECC) will be presented since it contains the bug triggering. In the class `SymVarSSA` we declared a symbolic boolean variable *confidential* and defined its getter and setter. We used it to set the return value of the function call `getenv("PATH")` to confidential. Thus, it is possible to specify other types of variables (sensitive, etc.) and expressions, which could be tainted.

The class `ModelGetEnvironment` (MGE) contained in Fig. 3 implements the `IFunctionModel` interface. The MGE sink model of the `getenv("PATH")` contains the implementations of the `exec()` and `getSignature()` methods. The `exec()` method will be called by the `Interpreter` in order to get the return value of the `getenv()`.

In getSignature() the parameters of the getenv() are defined and a return type is set. In the exec() method we taint the confidential value to be the return value of the exec() method. The IECC will be attached to the Interpreter in the class SymBolicExecutionEngine that contains the main program loop, which iterates through all program paths. During loop iteration when the Interpreter reaches the printf() C function or other trust-boundaries then the exec() method contained in the sink function model ModelPrinf will be called. Then the Interpreter calls the updateChecker() method which notifies the IE checker contained in IECC.

4. CHECKER IMPLEMENTATION

Based on a runtime language interpreter we are handling symbolic variables during static execution. Our statement processor enforces inference rules on C/C++ statements as the statement AST is traversed. For each node contained in the new path a statement processor instance will be instantiated. The inference information is constructed for each statement on the fly by enforcing inference rules based on explicit IF's.

When the static analysis is discovering that a confidential variable is about to pass through a previously defined trust-boundary then an IE bug report is issued which is reported in the Problems view inside the second Eclipse CDT instance. By clicking on the bug report available in the Problems view the user navigates to the bug location in the file where the bug was discovered. A bug report is composed of the file and line number where the bug was detected.

In order to propagate the confidential return value from the getenv() source to the printf() sink we had to extend the StatementProcessor (SP) class. A Interpreter object is instantiated for each new path. A new SP object will be instantiated by the Interpreter for every IASTNode (IbasicBlock) contained in the current path. Thus, propagating only the symbolic variables belonging to one execution path at a time. For each IASTNode the corresponding leave() methods are called depending on the type of the node. The leave() methods are used to traverse each statement AST in a bottom-up fashion. The leave() methods are also used for confidential variables propagation. The SP extends the ASTVisitor class which is an implementation of the visitor design pattern providing top-down (visit() methods) and bottom-up (leave() methods) traversal of each node contained in the current path. Each IASTNode is a C/C++ line (no comment lines are included) originating from the C/C++ test program file. When the SP detects that a symbolic variable or function return variable is confidential as each statement is traversed on the current path it tries to propagate the confidential variable based on explicit IF. For the CWE-526 test programs MGE is the source because from here confidential information flows into the program and ModelPrintf is the sink because here the potential information is leaving the program. When the SP detects the getenv() function inside the wrapper function printLine() then it adds a new confidential variable in the Interpreter. The confidential return value comes from MGE, which is the function model of the getenv() function call. The confidential variable is propagated to printline() as parameter. When the SP reaches the printline() statement a binding call is made. The binding call returns the parameters names of the printline() header function. The new parameter names are needed because these are used inside the printLine() implementation. These parameter names are potential confidential symbolic variables.

The printLine() function header has line as parameter. After we detect line in the method header and we know that we are on a potentially reachable path we add a new confidential variable called line in the Interpreter. This means that on this path from the source printLine(getenv("PATH")) to the sink printf("%s\n", line) the getenv("PATH") confidential return value will be assigned to the variable line which becomes also confidential. This happens when printLine() calls the execute() method. The implementation contains printf(), as presented in Fig. 1b. The SP proceeds until it reaches the printf("%s\n", line) node. After reaching this statement the Interpreter will be notified from the function model ModelPrintf using line as parameter. The interpreter will be notified because the statement printf("%s\n", line) is a sink. The Interpreter will be called with resolveOrigSymVar() and directly afterwards the getCurrentSSACopy() method will be called. These methods search in the Interpreter for a symbolic variable called line. After this call we get a SymPointerSSA variable s that we send over to the Interpreter by calling ps.notifyTrustBoundary(s). The Interpreter then calls our previously attached IF checker by calling his updateChecker(SymVarSSA s, IFile file, IASTFileLocation loc) method. The Interpreter sends to the IF checker the previously found s variable, the file and the location in the file from where it was notified. The IEC checks in the updateChecker() method if s is confidential. If s is confidential then a new a bug report will be created. For the test programs contained in CWE-534 and CWE-535 the propagation is similar to what we previously presented only the sinks and sources are different.

4.1 Tainting and Triggering

The implementation of the static analysis engine [31] is based on function models used for behavior description of standard C/C++ library function calls. A function model class contains 5 methods and implements the interface IFctModel.

First, the constructor, second the method getName() which returns the name of the function, third, getLibrarySignature() returns the whole function header as it is defined in the C standard library, fourth, exec(SymFunctionCall call) which is used for static execution of function calls (variables can be here tainted (Fig. 4, line 25) and trust boundaries used for notifying a checker) fifth, getSignature() returns a SymFctSignature object containing the data types of the function parameters and the return type of the function. The difference between the printf(), sink function model and the getenv() source function model is that in the exec() method of the printf() class we notify our IF checker that a trust-boundary is about to be passed and in the exec() method of the getenv() model we set the return value to confidential. Similarly it is implemented for the sinks and sources contained in the CWE-534/535 test programs.

```
0.  private Interpreter ps;
1.  public Mgetenv(Interpreter ps) {
2.  this.ps = ps;
3.  }
4.  public String getName() {
5.  return "getenv";
6.  }
7.  public SymFunctionReturn exec
8.    (SymFunctionCall call) {
9.      ArrayList<IName> plist = call.getParams();
```

```
10.      SymPointerOrig isp =
          ps.getLocalOrigSymPointer(plist.get(0));
11.      IName nebn = new EnvVarName();
12.      SymIntOrig sb_size = new SymIntOrig(new
          ImpVarName());
13.      SymArrayOrig sb = new SymArrayOrig(nebn,
          sb_size);
14.      SymPointerSSA isp_ssa = null;
15.    try {
16.      sb.setElemType(eSymType.SymPointer);
17.      ps.declareLocal(sb);
18.      ps.declareLocal(sb_size);
19.      SymArraySSA sb_ssa = (SymArraySSA)
20.      ps.getLocalOrigSymArray(nebn).
21.      getCurrentSSACopy();
22.      isp_ssa = (SymPointerSSA)
23.      ps.ssaCopy(isp);
24.     isp_ssa.setTargetType(eSymType.SymPointer);
25.     isp_ssa.setConfidential(true);
26.     isp_ssa.setTarget(sb);
27.    } catch (Exception e){
28.  e.printStackTrace();
29.  }
30.   return new SymFunctionReturn(isp_ssa);
31.  public SymFctSignature getSignature() {
32.    SymFctSignature fsign = new
          SymFctSignature();
33.    fsign.addParam(new  SymPointerOrig
          (eSymType.SymArray, new Integer(1)));
34.    fsign.setRType(new SymPointerOrig
35.      (eSymType.SymPointer, new Integer(1)));
36.   return fsign;
37.  }
```

Figure 4. The getenv() function model.

The SAE currently contains function models for the following C functions: `atoi()`, `fclose()`, `fgets()`, `fwgets()`, `fgetws()`, `fopen()`, `gets()`, `memcpy()`, `mod()`, `puts()`, `rand()`, `srand`, `strcpy()`, `strlen()`, `time()`, `wcscpy()`, `wcslen()`. For the IE test programs the following function models were added: CWE-526 `getenv()` (source), `printf()`(sink), CWE-534 and CWE-535 `LogonUserA()`, `LogonUserW()`(sources), `fprintf`, `fwprintf()`(sinks).

The models are used either to taint a symbolic variable with the tag *confidential* or to notify the IF checker that a confidential tagged variable is about to pass a trust-boundary.

4.2 Potential Implementation Gain

[33] presents some speedups through usage of backtracking inside the SAE and reports the performance increases in [31]. [33] reports a speedup of 5x-10x for statically analyzing 5978 SLOC representing a stack based buffer overflow, char type overrun memcpy() (CWE-121) and an speedup of 2x for 16567–SLOC representing a stack based buffer overflow, CWE-129 fgets() (CWE-121) on a Core 2 Quad CPU Q9550, on 64-bit Linux kernel 3.2.0.

We could use the new SAE version right of the box or implement our own interface which executes potential buggy paths in a backtracking manner. Compared from an architectural point of view by switching to a backtracking execution speedups of around 20 could be achieved for the test cases CWE-526/534/535.

Other implementation gains could be achieved by using state cloning, parallelization or path merging. Thus, further necessary experiments are necessary in the future in order to prove our assumptions.

5. TOOL DEMO

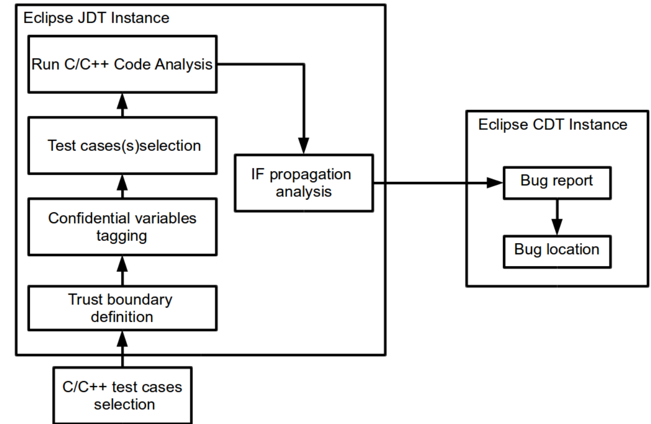

Figure 5. Information exposure checker work-flow.

The explicit IF theory for propagating confidential variables from trust-boundarys (sources) or other program points to trust-boundaries (sinks) was used in this paper. Our information-flow checker is based on the SAE that proved to scale for other types of bug checkers and larger test cases as well. The work flow used for running our checker is presented in Fig. 5. First, the C/C++ test programs have to be selected and test programs created in the workspace. Second, the trust boundaries have to be defined and confidential variables need to be tainted. Third, one or more test programs available in the workspace can be selected and the sub-menu button *Run C/C++ Code Analysis* needs to be selected.

The IE checker runs as an Eclipse plug-in project. The checker is launched as a standard Eclipse application. After starting the checker a second Eclipse CDT instance will be launched. The new Eclipse instance is presented in Fig. 6. For the test cases CWE-526/534/535 we had 90 Test Programs (TPr) contained each in an separate Eclipse CDT project. The TPr's don't have to be executable in order for us to perform static analysis. We run our checker by right-click on the 16th project for example and selecting *Run C/C++ Code Analysis* as highlighted in Fig. 6 with the mouse pointer. The sub-menu presented in Fig. 6 appears by clicking right on one or more selected Eclipse CDT projects.

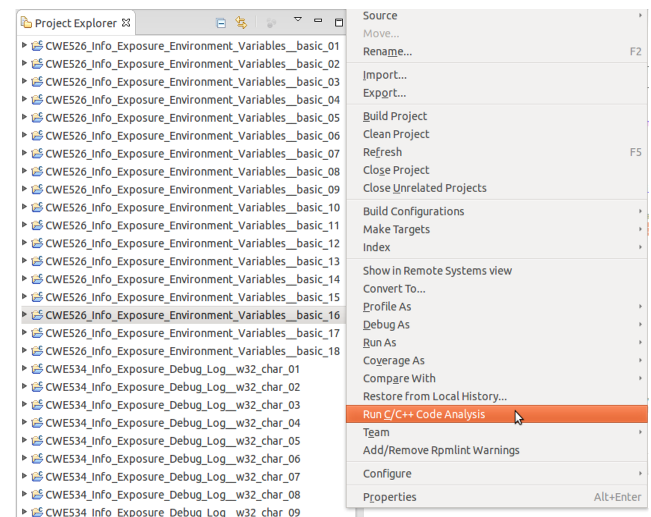

Figure 6. Triggering the IF checker from the Codan GUI.

The Codan API [34] provides a Graphical User Interface (GUI) for running checkers. The result of the execution of the checker can be observed in Fig. 7. The numbering from ① to ③ in Fig. 7 highlights the main GUI features available when starting the IE checker. Number ① indicates for which project the IE checker was started. Number ③ indicates the location where the bug has been detected. In Fig. 7 number ③ indicates with an bug icon that at line 13 a buggy statement was detected. Also the whole statement where the bug was detected will be highlighted with an underlining zigzag line. Another Codan API feature used for displaying bug reports is represented by the possibility to configure bug reports as Warnings, Errors or Infos, as presented in Fig. 8b.

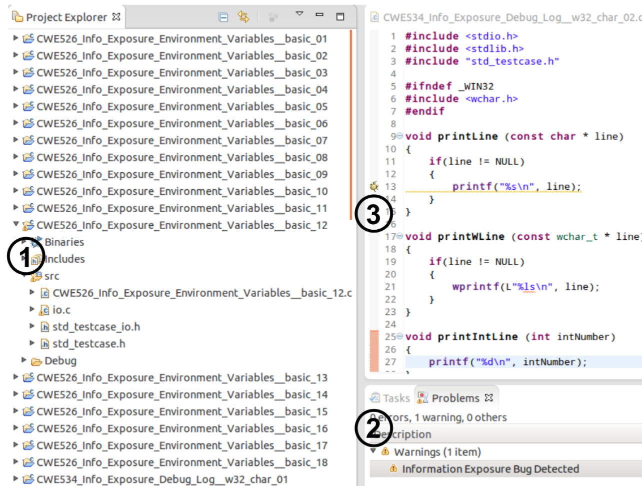

Figure 7. IF checker bug report and bug highlighting.

For the bug report presented in Fig. 7 with number ② we get the Description (containing a string which the user can configure), Resource (the file where the bug has appeared), Path (path of the file in the project hierarchy were the bug has appeared), Location (the line where the bug was reported) and Type (the type of the reported bug).

The output of the IF checker is a bug report for each detected IE bug. By double clicking on ② the user can navigate in the file at the line number where the bug was detected. One such bug report for the test program 16 contained in the test case CWE-526 is highlighted in Fig. 7 with number ② and the file location (file name and line number) of the bug with number ③.

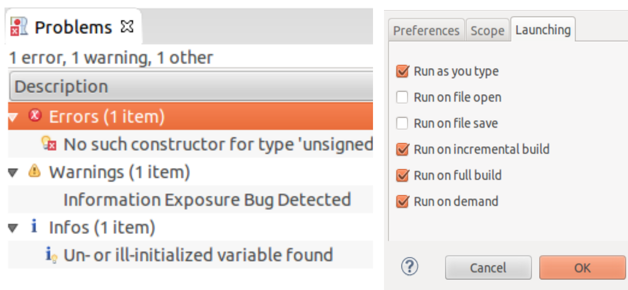

Figure 8a. Codan report types. **Figure 8b. IF checker running modes.**

Fig. 8a presents bug reports in a tree based view where every bug is classified based on one of the following three categories: Warnings (yellow triangle icon), Errors (red circle icon) or Infos (blue "i" symbol icon). The warning (Information Exposure Bug) presented in Fig. 8a corresponds to the bug report presented in Fig. 7. The "Errors" and "Infos" reports presented in Fig. 8a are not related to our IE checker. Codan reports use three different bug icons.

By clicking on the generated Information Exposure Bug report presented in Fig. 8a the appropriate file containing the bug is opened in the main view and the mouse cursor will point to the line number containing the bug as presented in Fig. 7, number ③.

The Codan API offers the possibility to configure each checker to be launched in different modes as presented in Fig. 8b. This bug triggering features can help an developer to control how and when Eclipse will trigger the bug detection analysis. Thus, helping to avoid bug insertion during software development.

6. EMPIRICAL EVALUATION

The goal of our empirical evaluation is to assess the efficiency of our IF checker in terms of number of detected false-negatives, false-positives, true-positives and execution time. At the same time we want to highlight to what extent our research work is significant. The evaluation was performed using the IE checker presented in Section 4 and the Juliet test cases CWE-524/534/535. Evaluation results are presented in Table 1.

6.1 Methodology

The test cases CWE-526/534/535 were selected because they contain information exposure bugs which we want to detect. These test cases are publicly available in the last version of the Juliet test suite [7] as of June 2014. CWE-526 contains 18 Test Programs (TPr), CWE-534 contains 36 TP, and CWE-535 contains 36 TP. For all the test programs contained in CWE-526/534/535 we created a separate Eclipse CDT project. The test programs were then inserted in one Eclipse workspace. In Fig. 7 some of the analyzed test programs can be observed.

The IE checker was run automatically for each test program available in the workspace by selecting once all the Eclipse CDT projects available in the workspace and selecting the sub-menu *Run C/C++ Code Analysis*. We measured the time from the moment of clicking the sub-menu button until all the projects in the workspace were completely analyzed. We also measured the execution time for the test programs belonging to one test case. We reported the intermediate execution time (for test programs belonging to one test case), total execution time, number of true-positives, false-negative and false-positives in Table 1.

For measuring the time between the moment when the analysis was started and the moment when all the test programs in the workspace were analyzed we used the following time stopping criteria. For determining the total execution time we monitored the event when there was no longer output messages in the console. For determining the intermediate execution times for test programs belonging to one of the test cases CWE-526/534/535 we monitored when all the test programs had a bug icon attached to them.

We new in advance which test programs should have an bug icon attached to them after running the static analysis and which test programs should not have a bug icon (for 5 out of 90 test programs it was not possible to perform the static analysis, this is reported in the next section) attached from previous runs. One such bug icon is presented in Fig. 7 above number ① and represents a yellow triangle with an exclamation mark inside.

6.2 Results and Constraints

Table 1, contains the results obtained by analyzing CWE-526, CWE-534 and CWE-535 with our IE checker. Table 1. contains

the following abbreviations: Test Program (TPr), Source Lines of Code (SLOC) without comments, FP (False-Positives), FN (False-Negatives), True Positives (TP) no programs containing the C `goto` statement included, Total True-Positives (TTP) per Test Case (TC), all programs included and Total Execution Time in Seconds (TES[s]) per TC. Used system: Ubuntu 12.04 LTS, Kernel 3.8.0-35-generic, 64-bit, Intel® Core™ i7-4770 CPU @ 3.40GHz × 8, 16 GB RAM.

Table 1. IF checker run-time results

Test case	TPr	SLOC	FP	FN	TP	TTP	TES[s]
CWE-526	18	5371	0	0	17	18	30
CWE-534	36	14876	0	0	34	36	46
CWE-535	36	14362	0	0	34	36	45
Total	90	34609	0	0	85	90	121

Our tool found 85 TP out of 90 TP present in the used test cases. We were able to detect all IE bugs. It was not possible to test all test programs available in the test cases because the Codan API is not supporting the building of the CFG for source code containing C `goto` statements, e.g. `goto stop;`.
The test cases CWE-526/534/535 contain 1, 2 and respectively 2 test programs containing the C `goto` statement. In total 5 out of 90 test programs were not analyzable. Thus, we reached 94.44% test coverage. We think that if this limitation will be removed from Codan API releases then 100% test coverage is achievable.

6.3 Comparison with Other Tools

Unfortunately we were not able to find any open source tool similar to our IE checker. We searched through related research and also the list provided by NIST [58] containing static analysis tools.
The list provided by NIST contains 53 tools from which 21 are free tools. From these 21 free tools only 11 tools can analyze C or C++ code. From the 11 tools only 3 tools can be used for detecting information-flow vulnerabilities.
Cqual, FlawFinder and Splint were used to analyze our test programs. None of the tools is based on SMT solvers and none was capable to detect the IF vulnerabilities present in the original test programs contained in CWE-526, CWE-534 and CWE-535.

6.4 Threats to Validity

Threats to external validity concern our ability to generalize the results of our empirical evaluation. In our empirical evaluation we have used three test cases, which were composed of C files having in total 90 test programs. Still there are a wide range of factors concerning testing platform and empirical evaluation methodology that may impact the test results.
Our IE checker is out of the box usable for other test cases as well. If the function models are not available in SAE then: first, the required function models need to be defined and second, the confidential variables need to be tainted. After this steps we can run our IE checker fully automatically by one mouse click.
We are aware that the number of function models is currently limited but the steps needed to define a function model follows the same design pattern for all the function models. Each function model contains 5 methods. In some cases almost all the source code of other available function models could be reused. Thus, offering a high level of code reuse.
Threats to internal validity concern our ability to draw conclusions about independent and dependent experimental conditions that make a difference or not if these are altered and whether there is sufficient evidence to support our claim. Regarding the definition of sinks and sources, which can cause IE leaks our own assessment may be subject to errors, incompetence or bias for test programs containing many KLOC. For small test programs as those used in this paper the definition of sinks and sources is not as error prone as for large test programs.
Threats to construct validity are concerned with how we used the Juliet test cases. All the used test cases are publicly available in the Juliet test suite. We created for each TP a separate Eclipse CDT project. We didn't want to modify the test programs in any form this is why we chosen this approach.
We think that neither of these “constraints” pose a real threat to the internal, external or construct validity of our approach as we provide also alternatives and argument why certain implementation decisions where made and others not.

7. RELATED WORK

Every kind of static taint analysis is based on a type of formalization [35]. Program dependence graphs [36], [37], program slicing techniques [38], or types systems [39] are used as in the CQual tool [40].
For scalability reasons, internal representations as Static Single Assignment (SSA), Gated Single Assignment (GSA) or Augmented Single Static Assignment (aSSA) are used [41].
During symbolic execution we use per path scope of symbolic variables.
We check for potential IE bugs only for reachable paths by querying the path validation. The path validation decides based on queries submitted to the MathSat SMT solver if the current path is satisfiable or not.
Deciding during static execution if a path is reachable reduces computational overhead and the total number of potential paths on which IE bugs could be located. Tainting confidential variables is done statically in the function models which are used to model each trust-boundary. Taint variable propagation is based on explicit IF.
Thus, a large amount of research work has been already published concerning symbolic variables tainting and propagating their values using static, dynamic or hybrid taint-analysis.
We briefly review in this section the most commonly used approaches focusing more on the works that are close to the one we proposed in this paper.

7.1 Static Taint-Analysis

One of the approaches to compute variable taintness is to use Static Taint Analysis (STA) techniques, allowing taking into account all possible execution paths. STA does not provide runtime information and environment interaction has to be simulated. Thus, the environment model introduces imperfections because it can not capture each real world interaction.
The majority of static taint analysis tools are based on user input dependencies [42]. Our tool handles each potential input source independently by modeling it with function models that simulate their execution during static execution. We are capable to model inputs from users, files, sockets and input streams in this way. Our tool is similar to the compile time analyzer PREfix [43] in the sense that both tools sequentially are tracing distinct execution paths and simulate the action of each operator and function call on the path.
Static taint-analysis can be used to enforce privacy control on mobile devices. Xiao et al. [20]propose a transparent privacy control approach that uses static symbolic execution [5] based on implicit IF's. Data is taint using scripts developed with the TouchDevelop [44], which allows users to create applications using an imperative, and statically typed language. Variable tainting is based on the fact that the whole TouchDevelop API on

which the user scripts are based is in advance taint with information concerning sources and sinks. This approach is different from ours because it supposes a previously known API where everything is taint and no other untainted sources or sinks exist. We define our sinks and sources directly in the function models and simulate real execution using them. Our approach can handle the definition of sinks and sources for the same procedure. Thus, introducing more flexibility during analysis.

Guarnieri et al. [17] taint variables using a central knowledge base. The authors propose an Eclipse based tool capable to detect IF vulnerabilities due to missing input validation with the help of a decentralized knowledge base and on AST's generated by the JDT compiler. The static analyzer detects security issues related to input validation problems in web applications. The IF analysis does not consider context sensitivity and it is not using an SMT solver. The framework offers the possibility to taint classes, packages or methods as trusted. The problem of IF vulnerability detection translates to identifications of errors between entry (sources) and exit points (sinks) that do not use a trusted object.

FindBugs [45], [46], [47] is based on Eclipse and it does not detect IE bugs but it uses the concept of easy integration of new checkers into the static analysis. It checks for bugs in Java byte code based on currently 300 patterns of coding mistakes for Java byte code. It employs intra-procedural analysis that takes into account information from instance of tests. FindBugs has a plug-in architecture in which detectors (code checkers) can be defined reporting different bug patterns. Detectors can access information about types, constant values, special flags and values stored on the stack or local variables. Some of the defined detectors perform intra-procedural summary information. FindBugs doesn't use SMT [29] or a SAT [30] solver in order to perform the static analysis but is rather based on the previously mentioned patterns, which can be extended by the so-called detector concept. Which is similar to our checker concept of easily attaching checkers to the language interpreter.

7.2 Dynamic Taint-Analysis

Another approach to computer variable taintness is based on Dynamic Taint Analysis (DTA), meaning that concrete program execution is performed. The main advantage of DTA is the possibility to use data flow information available during runtime but only from one path of execution at a time. Thus sanity checks can be handled accurately avoiding many false positives. However, since each analysis is reduced to a single (current) execution path, its coverage level may remain very weak and control dependencies cannot be fully taken into account. At the same time it cannot guarantee that all possible execution paths are exercised. Thus, it is in general geared towards explicit IF's.

The notion of taint variable was introduced with the Perl scripting language and its taint running mode where taint variables are propagated using the language interpreter across variable assignment and security errors are raised when an insecure system call appears. There is a wide range of proposed tools until now which are based on language information-flow security: Java-based JFlow [48] with its software tool Jif [49] developed an annotation language for Java code. Data values are labeled using security policies. The attached labels restrict the movement of data values thus enforcing a policy on the data flow. The programming languages: Caml-based FlowCaml [50] and Ada-based SPARK Examiner [51] and the scripting languages Perl, PHP, Ruby and Python have a taint mode similar to the taint mode available in Perl.

Dynamic taint analysis is not suitable for us because we want to have high path coverage and exercise all possible execution paths. Thus, we rely on a SMT solver, which helps to detect satisfiable paths and afterward it provides candidate paths to our IE checker.

7.3 Hybrid Taint-Analysis

Hybrid taint-analysis is a combination of the previously two mentioned approaches. Hybrid taint-analysis approaches benefit from having during static analysis information available from dynamic execution or vice-verse (during dynamic analysis information from static execution). Thus, overcoming some shortcomings of both.

The first approach explores executable paths in the same way as static symbolic execution does and interleaves concrete execution with symbolic execution. Concrete values from execution are used by these techniques when difficult constrains are reached allowing the algorithm to proceed. Concolic testing [52] is one of the most prominent hybrid analysis. Concolic techniques can reason precisely about complex data structures and simplify constrains when they exceed the capabilities of the solver. KLEE [13] is a Concolic testing tool for C programs which extends EXE [53] and addresses path explosion by allowing interaction with the outside environment without using entirely concrete procedure call arguments.

The second approach is mostly based on IF monitors which monitor the execution of the program and use information from static analysis to decide for example when it is safe to stop tracking of confidential variables. Moore et al. [21] propose a hybrid IF monitor, which combines static analysis and dynamic mechanisms in order to provide strong information security guarantees. Their approach adds runtime overhead in comparison to pure static analysis. The authors argue that their static analysis can determine when it is sound for a monitor to stop tracking the security level of certain variables. This extra information is provided through the usage of static analysis, which can reason in general more precise about certain IF's [24] proved by Russo and Sabelfeld. Their implementation extends the information-flow monitor of Russo and Sabelfeld [24].

The authors present conditions for incorporating memory abstractions and analysis into a hybrid information-flow monitor. The static analysis relies on a flow-sensitive security type system [54] which helps to determine when a variable cannot cause a security violation.

The environment taints each program variable with a security level and tracks the currently stored security level. Thus, allowing for a kind of automatic tainting and propagation of taint variables whereas we initially taint variables statically in our function models.

To the best of our knowledge our checker is the only IE bug checker that uses symbolic execution to find potential candidate paths on which IE bugs could reside and that propagates confidential variables based on explicit IF's.

8. CONCLUSION AND FUTURE WORK

We successfully proved that our IE checker can be used for detecting IE bugs and at the same time we have shown to what extent our work is significant by comparing it with related research work.

The Codan API was used for parsing source files, dealing with project resources and interpreting C/C++ code. Bug location marking was easily implemented using the markup capabilities offered by the Codan API.

The AST traversing mechanisms offered the possibility to focus on more on static analysis and not on re-implementing utility functions for manipulating AST nodes. Building of CFG for test programs containing C `goto` statements should be possible in future Codan API releases. Thus, removing one of the current implementation constrains.

We successfully used the SAE engine for propagating symbolic variables tags based on explicit information flow. The SAE engine

was easily extendable and offered the possibility of plugging our IE checker in the existing language interpreter.

The computational overhead introduced by SMT-lib [55] statement construction and the calling of an external command line tool could be avoided by using an SMT solver, which has an API compatible with our development language [56] or [57].

The static definition of sinks, sources and confidential variables can be automated in the following ways. First, if we previously attach extra information to methods and attributes in an UML class diagram and then generate code from this model, Code 2 Model (C2M). Second, by converting a test program into a UML class diagram and annotate it with security annotations inside a special designed tool for this purpose, C2M could help a security expert to design secure software systems. Afterward source code can be generated that contains code assertions added in the previous step, Model 2 Code (M2C). These assertions could be used to identify sinks, sources and confidential variables automatically. Utility classes for our checkers can be generated and also we could profit from multiplication of assertions in the source code due to M2C conversion. Third, the definition of sinks, sources and confidential variables can be automated by providing annotated test cases or libraries containing this information or by loading a configuration file containing the specifications.

In future we plan to do research in the area of annotating whole C/C++ libraries and reusing these annotated libraries during static analysis for trust-boundarys definition and symbolic variable tainting. We envisage that advance checks for sinks and sources on a potential bug prone path could reduce the number of candidate paths.

The process of manually determining which function models (trust-boundaries) could produce IE leaks could be automated by preprocessing the source code and determining which functions could represent potential candidates for sinks and sources. This could be based on a previously annotated C/C++ library with annotation tags attached to function headers. Information propagation could be used in order to detect other potential sink, sources, etc. Thus, fully automating the process of trust-boundary definition and initial variable tainting.

We think that static analysis is worthwhile to be used for detecting IE bugs related to security concerns and future research in this area is needed.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. This research is funded by the German Ministry for Education and Research (BMBF) under grant number 01IS13020.

10. REFERENCES

[1] S. Zdancewic and A.C. Myers, "Robust Declassification", In Proceedings of the IEEE Computer Security Foundations Workshop, pp. 15–23, 2011.

[2] http://cwe.mitre.org/data/definitions/200.html, accessed on June 2014.

[3] https://www.owasp.org/index.php/Top_10_2007, accessed on June 2014.

[4] http://blog.veracode.com/2010/12/mobile-app-top-10-list/, accessed on June 2014.

[5] James C. King, "Symbolic Execution and Program Testing", Commun. ACM, Vol. 19, pp. 385–394, 1976.

[6] D.S. Kushwana and A.K. Misra, "Software test effort estimation", ACM SIGSOFT Software Engineering Notes archive Vol. 33 Issue 3, Article No. 6, May 2008.

[7] United States, National Institute of Standards and Technology (NIST): Juliet Test Suite v1.2 for C/C++, online: http://samate.nist.gov/SRD/testsuites/juliet/Juliet_Test_Suite_v1.2_for_C_Cpp.zip, accessed on June 2014.

[8] M. Das, S. Lerner and M. Seigle, "ESP:path-sensitive program verification in polynomial time", PLDI '02, ACM SIGPLAN Conference on Programming language design and implementation, pp. 57 – 68, 2002.

[9] T. Ball and S.K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces", Proc. 8th SPIN Workshop on Model Checking of Software, LNCS 2057, Springer-Verlag, pp.103–122, 2001.

[10] T. A. Henzinger, R. Jhala, R.Majumdar and G. Sutre, "SoftwareVerification with Blast", Proc. 10th In. Workshop Model Checking of Software, LNCS 2648, Springer-Verlag, pp. 235–239, 2003.

[11] FindBugs, http://findbugs.sourceforge.net, accessed on Jan. 2014.

[12] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later ", Communications of the ACM (CACM) Volume 56, Issue 2, 2013.

[13] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs", In OSDI'08, Dec. 2008.

[14] J. S. Fenton, "Memoryless subsystems", Computer Journal, vol. 17, no. 2, pp. 143–147, May 1974.

[15] A. Sabelfeld and A. Russo, "From dynamic to static and back: Riding the roller coaster of information-flow control research", in Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics, pp. 352–365, 2009.

[16] T. Avgerinos, S.K. Cha, B. L. T. Hao and D. Brumley, "AEG: Automatic Exploit Generation", In Proceedings of the Network and Distributed System Security Symposium (NDSS 11), San Diego, CA, Feb. 2011.

[17] M. Guarnieri, P. El Khoury and G. Serme, "Security vulnerabilities detection and protection using Eclipse", ECLIPSE-IT 2011, 6th Workshop of the Italian Eclipse Community, Milano, Italy, September 22-23, 2011.

[18] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis", Journal of Computer Security, vol. 4, no. 3, pp. 167–187, 1996.

[19] V. Simonet, "The Flow Caml System: documentation and user's manual", Institut National de Recherche en Informatique et en Automatique (INRIA), Technical Report 0282, Jul. 2003.

[20] X. Xiao, N. Tillmann, M. Fahndrich, J. de Halleux and M. Moskal, "Transparent Privacy Control via Static Information Flow Analysis", Microsoft Research Tech Report MSR-TR-2011-93, Aug. 7, 2011.

[21] S. Moore and S. Chong, "Static analysis for efficient hybrid information-flow control", CSF '11 Proceedings of the IEEE 24th Computer Security Foundations Symposium pp. 146-160, 2011.

[22] E. S. Cohen, "Information transmission in sequential programs", In R. A. DeMillo, D. P. Dobkin, A.K. Jones and R. J. Lipton, editors, Foundations of Secure Computation, Academic Press, pp. 297-335. 1978.

[23] D. E. Denning and P. J. Denning, “Certification of programs for secure information flow”, Comm. of the ACM, 20(7):504–513, July 1977.

[24] A. Russo and A. Sabelfeld, “Dynamic vs. Static Flow-Sensitive Security Analysis”, Proceeding CSF '10 Proceedings of the 23rd IEEE Computer Security Foundations Symposium, pp. 186-199, 2010.

[25] Tiobe index of programming languages, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, accessed on Jan. 2014.

[26] Langpop index of most used programming languages, http://langpop.com/ , accessed on Jan. 2014.

[27] IEEE Spectrum, Top 10 Programming Languages http://spectrum.ieee.org/computing/software/top-10-programming-languages , accessed on September 2014.

[28] http://zeroturnaround.com/rebellabs/using-eclipse-for-java-development/ , accessed on Jan. 2014

[29] Harrison, J., “Handbook of Practical Logic and Automated Reasoning”, Cambridge University Press, 2009.

[30] A. Armando, J. Mantovani, L. Platania, “Bounded model checking of software using SMT solvers instead of SAT solvers”. Int. J. Softw. Tools Technol. Transf.11(1), pp. 69–83, 2009.

[31] A. Ibing, “SMT-constrained symbolic execution for Eclipse CDT/Codan”, In: Workshop on Formal Methods in the Development of Software, 2013.

[32] A. Cimatti, A. Griggio, B. Schaafsma and R. Sebastiani, “The MathSAT 5 SMT solver” In: TACAS 2013.

[33] A. Ibing, “Parallel SMT-Constrained Symbolic Execution for Eclipse CDT/Codan”, 25th IFIP International Conference, ICTSS 2013, Lecture Notes in Computer Science, Istanbul, Turkey, pp. 196-206, Nov. 13-15, 2013.

[34] A. Laskavaia, “Codan- C/C++ static analysis framework for CDT”, In: EclipseCon 2011.

[35] D. Ceara, L. Mounier and M. L. Potet, “Taint Dependency Sequences: a characterization of insecure execution paths based”, ICSTW '10, pp. 371-380, 2010.

[36] C. Hammer, J. Krinke, and G. Snelting, “Information Flow Control for Java Based on Path Conditions in Dependence Graphs,” in In IEEE International Symposium on Secure Software Engineering, 2006.

[37] G. Snelting, T. Robschink, and J. Krinke, “Efficient path conditions in dependence graphs for software safety analysis”, ACM Trans. Softw. Eng. Methodol., vol. 15, no. 4, 2006.

[38] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar, “Interprocedural analysis for privileged code placement and tainted variable detection,” in ECOOP 2005 - Object-Oriented Programming, pp. 362–386, July 2005.

[39] J. S. Foster, T. Terauchi, and A. Aiken, “Flow-sensitive type qualifiers,” in PLDI ’02: Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation, pp. 1–12, 2002.

[40] http://www.cs.umd.edu/ jfoster/cqual/., accessed Feb. 2014.

[41] B. Scholz, C. Zhang, and C. Cifuentes, “User-input dependence analysis via graph reachability”, in Source Code Analysis and Manipulation, IEEE International Workshop on, Los Alamitos, CA, USA, pp. 25–34, 2008.

[42] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov, “Inputs of Coma: Static Detection of Denial of-Service Vulnerabilities,” in CSF ’09: Proceedings of the 22nd IEEE Computer Security Foundations Symposium. Washington, DC, USA, pp. 186–199, 2009.

[43] W. R. Bush, J. D. Pincus and D. J. Sielaff Journal, “A static analyzer for finding dynamic programming errors”, Software —Practice & Experience archive Volume 30 Issue 7, pp. 775-802, June 2000.

[44] N. Tillmann, M. Moskal, and J. de Halleux, “TouchStudio - Programming Cloud-Connected Mobile Devices via Touchscreen”, Microsoft Technical Report MSR-TR-2011-49, 2011.

[45] D. Hovemeyer and W. Pugh,“Finding Bugs is Easy”, ACM OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications , pp. 132-136, 2004.

[46] N. Ayewah and W. Pugh, “The Google FindBugs Fixit”, In Proceedings of ISSTA, Trento, Italy, pp. 241-252, July 12-16, 2010.

[47] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix,and W.Pugh, ”Experiences Using Static Analysis to Find Bugs”, Journal IEEE Software archive Vol. 25 Issue 5, pp. 22-29, Sep. 2008.

[48] A. C. Myers, “JFlow: Practical Mostly-Static Information Flow Control”, Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL ’99), San Antonio, Texas, USA, Jan. 1999

[49] A.C. Myers, L. Zheng, S. Zdancewic, S. Chong and N. Nystrom, Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001-2008.

[50] V. Simonet. The Flow Caml system. Software release. Located at http://cristal.intia.fr/~simonet/soft/flowcaml, July 2003.

[51] R. Chapman and A. Hilton, “Enforcing security and safety models with an information flow analysis tool”, ACM SIGAda, Ada Letters, 24(4):39–46, 2004.

[52] X. Qu and B. Robinson, “A Case Study of Concolic Testing Tools and their Limitations”, In: ESEM '11, IEEE Computer Society Washington, DC, USA, pp. 117-126, 2011.

[53] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill and D. Engler, “EXE: Automatically Generating Inputs of Death”, ACM Transactions on Information and System Security (TISSEC) Volume 12, No. 2, Dec. 2008.

[54] S. Hunt and D. Sands, “On flow-sensitive security types”, in Conference Record of the 33th Annual ACM, (POPL '06), pp. 79–90, 2006.

[55] C. Barrett, A. Stump, and C. Tinelli: The SMT-LIB Standard Version 2.0. (Dec. 2010), online http: //goedel. cs.uiowa. edu /smtlib/papers/ smt-lib-reference-v2.0-r10.12.21.pdf., accessed Jan. 2014.

[56] J. Christ, J. Hoenicke and A. Nutz, “SMTInterpol: an interpolating SMT solver”, SPIN'12, pp. 248-254, 2012.

[57] L. de Moura and N. Bjørner, “Z3: an efficient SMT solver”, TACAS'08/ETAPS'08 Proceedings of the Theory and practice of software, pp. 337-340.

[58] Source Code Security Analyzers, http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html, accessed on June 2014.